An Analytical Model for Multi-tier Internet Services and Its Applications

Bhuvan Urgaonkar, Giovanni Pacifici†, Prashant Shenoy, Mike Spreitzer†, and Asser Tantawi†

Dept. of Computer Science,
University of Massachusetts,
Amherst, MA 01003
{bhuvan,shenoy}@cs.umass.edu

† Service Management Middleware Dept.,
IBM T. J. Watson Research Center,
Hawthorne, NY 10532
{giovanni,mspreitz,tantawi}@us.ibm.com

ABSTRACT

Since many Internet applications employ a multi-tier architecture, in this paper, we focus on the problem of analytically modeling the behavior of such applications. We present a model based on a network of queues, where the queues represent different tiers of the application. Our model is sufficiently general to capture (i) the behavior of tiers with significantly different performance characteristics and (ii) application idiosyncrasies such as session-based workloads, tier replication, load imbalances across replicas, and caching at intermediate tiers. We validate our model using real multi-tier applications running on a Linux server cluster. Our experiments indicate that our model faithfully captures the performance of these applications for a number of workloads and configurations. For a variety of scenarios, including those with caching at one of the application tiers, the average response times predicted by our model were within the 95% confidence intervals of the observed average response times. Our experiments also demonstrate the utility of the model for dynamic capacity provisioning, performance prediction, bottleneck identification, and session policing. In one scenario, where the request arrival rate increased from less than 1500 to nearly 4200 requests/min, a dynamic provisioning technique employing our model was able to maintain response time targets by increasing the capacity of two of the application tiers by factors of 2 and 3.5, respectively.

1. INTRODUCTION

1.1 Motivation

Internet applications such as online news, retail, and financial sites have become commonplace in recent years. Modern Internet applications are complex software systems that employ a multi-tier architecture and are replicated or distributed on a cluster of servers. Each tier provides a certain functionality to its preceding tier and makes use of the functionality provided by its successor to carry out its part of the overall request processing. For instance, a typical e-commerce application consists of three tiers—a front-end Web tier that is responsible for HTTP processing, a middle tier Java enterprise server that implements core application functionality, and a back-end database that stores product catalogs and user orders. In this example, incoming requests undergo HTTP processing, processing by Java application server, and trigger queries or transactions at the database.

This paper focuses on analytically modeling the behavior of multi-tier Internet applications. Such a model is important for the following reasons: (i) *capacity provisioning,* which enables a server farm to determine how much capacity to allocate to an application in order for it to service its peak workload; (ii) *performance prediction,* which enables the response time of the application to be determined for a given workload and a given hardware and software configuration, (iii) *application configuration,* which enables various configuration parameters of the application to be determined for a certain performance goal, (iv) *bottleneck identification and tuning,* which enables system bottlenecks to be identified for purposes of tuning, and (v) *request policing,* which enables the application to turn away excess requests during transient overloads.

Modeling of single-tier applications such as vanilla Web servers (e.g., Apache) is well studied [4, 12, 17]. In contrast, modeling of multi-tier applications is less well studied, even though this flexible architecture is widely used for constructing Internet applications and services. Extending single-tier models to multi-tier scenarios is non-trivial due to the following reasons. First, various application tiers such as Web, Java, and database servers have vastly different performance characteristics and collectively modeling their behavior is a difficult task. Further, in a multi-tier application, (i) some tiers may be replicated while others are not, (ii) the replicas may not be perfectly load balanced, and (iii) caching may be employed at intermediate tiers—all of which complicate the performance modeling. Finally, modern Internet workloads are session-based, where each session comprises a sequence of requests with think-times in between. For instance, a session at an online retailer comprises the sequence of user requests to browse the product catalog and to make a purchase. Sessions are stateful from the perspective of the application, an aspect that must be incorporated into the model. The design of an analytical model that can capture the impact of these factors is the focus of this paper.

1.2 Research Contributions

This paper presents a model of a multi-tier Internet application based on a network of queues, where the queues represent different tiers of the application. Our model can handle applications with an *arbitrary* number of tiers and those with significantly different performance characteristics. A key contribution of our work is that

.

the complex task of modeling a multi-tier application is reduced to the modeling of request processing at individual tiers and the flow of requests across tiers. Our model is inherently designed to handle session-based workloads and can account for application idiosyncrasies such as replication at tiers, load imbalances across replicas, caching effects, and concurrency limits at each tier.

We validate the model using two open-source multi-tier applications running on a Linux-based server cluster. We demonstrate the ability of our model to accurately capture the effects of a number of commonly used techniques such as query caching at the database tier and class-based service differentiation. For a variety of scenarios, including an online auction application employing query caching at its database tier, the average response times predicted by our model were within the 95% confidence intervals of the observed average response times. We conduct a detailed experimental study using our prototype to demonstrate the utility of our model for the purposes of dynamic provisioning, response time prediction, application configuration, and request policing. Our experiments demonstrate the ability of our model to correctly identify bottlenecks in the system and the shifting of bottlenecks due to variations in the Internet workload. In one scenario, where the arrival rate to an application increased from 1500 to nearly 4200 requests/min, our model was able to continue meeting response time targets by successfully identifying the two bottleneck tiers and increasing their capacity by factors of 2 and 3.5, respectively.

The remainder of this paper is structured as follows. Section 2 provides an overview of multi-tier applications and related work. We describe our model in Sections 3 and 4. Sections 5 and 6 present experimental validation of the model and an illustration of its applications respectively. Finally, Section 7 presents our conclusions.

2. BACKGROUND AND RELATED WORK

This section provides an overview of multi-tier applications and the underlying server platform assumed in our work. We also discuss related work in the area.

2.1 Internet Application Architecture

Modern Internet applications are designed using multiple tiers (the terms Internet application and service are used interchangeably in this paper). A multi-tier architecture provides a flexible, modular approach for designing such applications. Each application tier provides certain functionality to its preceding tier and uses the functionality provided by its successor to carry out its part of the overall request processing. The various tiers participate in the processing of each incoming request during its lifetime in the system. Depending on the processing demand, a tier may be replicated using clustering techniques. In such an event, a dispatcher is used at each replicated tier to distribute requests among the replicas for the purpose of load balancing. Figure 1 depicts a three-tier application where the first two tiers are replicated, while the third one is not. Such an architecture is commonly employed by e-commerce applications where a clustered Web server and a clustered Java application server constitute the first two tiers, and the third tier consists of a non-replicable database.[1]

The workload of an Internet application is assumed to be session-based, where a session consists of a succession of requests issued by a client with think times in between. If a session is stateful, which is often the case, successive requests will need to be serviced by the
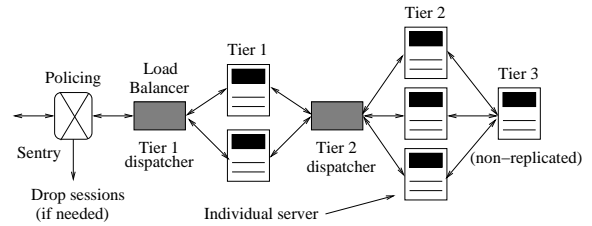
---

[1] Traditionally database servers have employed a *shared-nothing* architecture that does not support replication. However, certain new databases employ a shared-everything architecture [13] that supports clustering and replication but with certain constraints.

Figure 1: A three-tier application.

same server at each tier, and the dispatcher will need account for this server state when redirecting requests.

As shown in Figure 1, each application employs a sentry that polices incoming sessions to an application's server pool—incoming sessions are subjected to admission control at the sentry to ensure that the contracted performance guarantees are met; excess sessions are turned away during overloads.

We assume that Internet applications typically run on a server cluster that is commonly referred to as a data center. In this work, we assume that each tier of an application (or each replica of a tier) runs on a separate server. This is referred to as *dedicated hosting*, where each application runs on a subset of the servers and a server is allocated to *at most* one application tier at any given time. Unlike *shared hosting* where multiple small applications share each server, dedicated hosting is used for running large clustered applications where server sharing is infeasible due to the workload demand imposed on each individual application.

Given an Internet application, we assume that it specifies its desired performance requirement in the form of a service-level agreement (SLA). The SLA assumed in this work is a bound on the average response time that is acceptable to the application. For instance, the application SLA may specify that the average response time should not exceed one second regardless of the workload.

2.2 Request Processing in Multi-tier Applications

Consider a multi-tier application consisting of M tiers denoted by T_1, T_2 through T_M. In the simplest case, each request is processed exactly once by tier T_i and then forwarded to tier T_{i+1} for further processing. Once the result is computed by the final tier T_M, it is sent back to T_{M-1}, which processes this result and sends it to T_{M-2} and so on. Thus, the result is processed by each tier in the reverse order until it reaches T_1, which then sends it to the client. Figure 2 illustrates the steps involved in processing a "bid" request at a three-tier online auction site. The figure shows how the request trickles downstream and how the result propagates upstream through the various tiers.

More complex processing at the tiers is also possible. In such scenarios, each request can visit a tier *multiple* times. As an example, consider a keyword search at an online superstore, which triggers a query on the music catalog, a query on the book catalog and so on. These queries can be issued to the database tier sequentially, where each query is issued after the result of the previous query has been received, or in parallel. Thus, in the general case, each request at tier T_i can trigger multiple requests to tier T_{i+1}. In the sequential case, each of these requests is issued to T_{i+1} once the result of the previous request has finished. In the parallel case, all requests are issued to T_{i+1} at once. In both cases, all results are merged and then sent back to the upstream tier T_{i-1}.

2.3 Related Work

Modeling of single-tier Internet applications, of which HTTP servers are the most common example, has been studied extensively. A queuing model of a Web server serving static content was proposed in [17]. The model employs a network of four queues—two modeling the Web server itself, and the other two modeling the Internet communication network. A queuing model for performance prediction of single-tier Web servers with static content was proposed in [4]. This approach (i) explicitly models CPU, memory, and disk bandwidth in the Web server, (ii) utilizes knowledge of file size and popularity distributions, and (iii) relates average response time to available resources. A GPS-based queuing model of a single resource, such as the CPU, at a Web server was proposed in [3]. The model is parameterized by online measurements and is used to determine the resource allocation needed to meet desired average response time targets. A G/G/1 queuing model for replicated single-tier applications (e.g., clustered Web servers) has been proposed in [18]. The architecture and prototype implementation of a performance management system for cluster-based Web services was proposed in [11]. The work employs an M/M/1 queuing model to compute responses times of Web requests. A model of a Web server for the purpose of performance control using classical feedback control theory was studied in [1]; an implementation and evaluation using the Apache Web server was also presented in the work. A combination of a Markov chain model and a queuing network model to capture the operation of a Web server was presented in [12]—the former model represents the software architecture employed by the Web server (e.g. process-based versus thread-based) while the latter computes the Web server's throughput.

Since these efforts focus primarily on single-tier Web servers, they are not directly applicable to applications employing multiple tiers, or to components such as Java enterprise servers or database servers employed by multi-tier applications. Further, many of the above efforts assume static Web content, while multi-tier applications, by their very nature, serve dynamic Web content.

A few recent efforts have focused on the modeling of multi-tier applications. However, many of these efforts either make simplifying assumptions or are based on simple extensions of single-tier models. A number of papers have taken the approach of modeling only the *most constrained* or the *most bottlenecked* tier of the application. For instance, [19] considers the problem of provisioning servers for only the Java application tier; it uses an M/G/1/PS model for each server in this tier. Similarly, the Java application tier of an e-commerce application with N servers is modeled as a G/G/N queuing system in [14]. Other efforts have modeled the entire multi-tier application using a single queue—an example is [7], that uses a M/GI/1/PS model for an e-commerce application. While these approaches are useful for specific scenarios, they have many limitations. For instance, modeling only a single bottlenecked tier of a multi-tier application will fail to capture caching effects at other tiers. Such a model can not be used for capacity provisioning of other tiers. Finally, as we show in our experiments, system bottlenecks can shift from one tier to another with changes in workload characteristics. Under these scenarios, there is no single tier that is the "most constrained". In this paper, we present a model of a multi-tier application that overcomes these drawbacks. Our model explicitly accounts for the presence of all tiers and also captures application artifacts such as session-based workloads, tier replication, load imbalances, caching effects, and concurrency limits.

3. A MODEL FOR A MULTI-TIER INTERNET APPLICATION

In this section, we present a baseline queuing model for a multi-
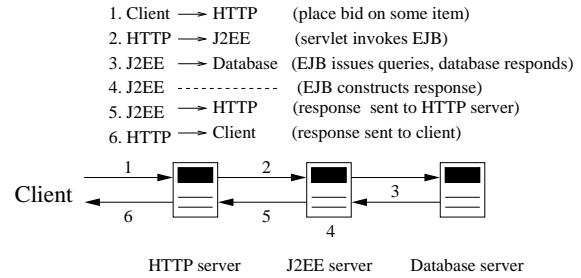

Figure 2: Request processing in an online auction application.

tier Internet application, followed by several enhancements to the model to capture certain application idiosyncrasies.

3.1 The Basic Queuing Model

Consider an application with M tiers denoted by $T_1, \cdots, T_M$. Initially we assume that no tier is replicated—each tier is assumed to run on exactly one server, an assumption that is relaxed later.

Modeling Multiple Tiers: We model the application using a network of of M queues, $\mathcal{Q}_1, \cdots, \mathcal{Q}_M$ (see Figure 3). Each queue represents an application tier and the underlying server that it runs on. We assume a processor sharing (PS) discipline at each queue, since it closely approximates the scheduling policies employed by most commodity operating systems (e.g., Linux CPU time-sharing).

When a request arrives at tier T_i it triggers one or more requests at its subsequent tier T_{i+1}; recall the example of a keyword search that triggers multiple queries at different product catalogs. In our queuing model, we can capture this phenomenon by allowing a request to make multiple *visits* to each of the queues during its overall execution. This is achieved by introducing a transition from each queue to its predecessor, as shown in Figure 3. A request, after some processing at queue $\mathcal{Q}_i$, either returns to $\mathcal{Q}_{i-1}$ with a certain probability p_i or proceeds to $\mathcal{Q}_{i+1}$ with probability $(1 - p_i)$. The only exceptions are the last tier queue $\mathcal{Q}_M$, where all requests return to the previous queue, and the first queue $\mathcal{Q}_1$, where a transition to the preceding queue denotes request completion. As argued in Section 3.2, our model can handle multiple visits to a tier regardless of whether they occur sequentially or in parallel.

Observe that caching effects are naturally captured by this model. If caching is employed at tier T_i, a cache hit causes the request to immediately return to the previous queue $\mathcal{Q}_{i-1}$ without triggering any work in queues $\mathcal{Q}_{i+1}$ or later. Thus, the impact of cache hits and misses can be incorporated by appropriately determining the transition probability p_i and the service time of a request at $\mathcal{Q}_i$.

Modeling Sessions: Recall from Section 2 that Internet workloads are session-based. A session issues one or more requests during its lifetime, one after another, with think times in between (we refer to this duration as the *user think time*). Typical sessions in an Internet application may last several minutes. Thus, our model needs to capture the relatively long-lived nature of sessions as well as the response times of individual requests within a session.

We do so by augmenting our queuing network with a subsystem modeling the active sessions of the application. We model sessions using an *infinite server queuing system*, $\mathcal{Q}_0$, that feeds our network of queues and forms the closed-queuing system shown in Figure 3. The servers in $\mathcal{Q}_0$ capture the session-based nature of the workload as follows. Each active session is assumed to "occupy" one server in $\mathcal{Q}_0$. As shown in Figure 3, a request issued by a session emanates from a server in $\mathcal{Q}_0$ and enters the application at $\mathcal{Q}_1$. It then moves through the queues $\mathcal{Q}_1, \cdots, \mathcal{Q}_M$, possibly visiting some queues
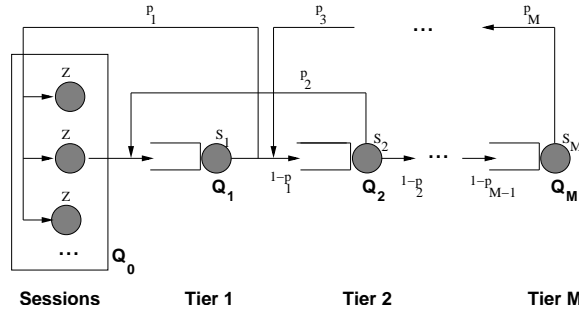
Figure 3: Modeling a multi-tier application using a network of queues.

multiple times (as captured by the transitions from each tier to its preceding tier) and getting processed at the visited queues. Eventually, its processing completes, and it returns to a server in $\mathcal{Q}_0$. The time spent at this server models the think time of the user; the next request of the session is issued subsequently. The infinite server system also enables the model to capture the independence of the user think times from the request service times at the application.

Let S_i denote the service time of a request at $\mathcal{Q}_i$ ($1 \leq i \leq M$). Also, p_i denotes the probability of a request making a transition from $\mathcal{Q}_i$ to $\mathcal{Q}_{i-1}$ (note that $p_M = 1$); p_1 denotes the probability of transition from $\mathcal{Q}_1$ to $\mathcal{Q}_0$. Finally, let Z denote the service time at any server in $\mathcal{Q}_0$ (which is essentially the user think time). Our model requires these parameters as inputs in order to compute the average end-to-end response time of a request.

Our discussion thus far has implicitly assumed that sessions never terminate. In practice, the number of sessions being serviced will vary as existing sessions terminate and new sessions arrive. Our model can compute the mean response time for a given number of concurrent sessions N. This property can be used for admission control at the application sentry, as discussed in Section 6.2.

3.2 Deriving Response Times From the Model

The Mean-Value Analysis (MVA) algorithm [15] for closed-queuing networks can be used to compute the mean response time experienced by a request in our network of queues. The MVA algorithm is based on the following key queuing theory result: *In product-form closed queuing networks[2], when a request moves from queue $\mathcal{Q}_i$ to another queue $\mathcal{Q}_j$, it sees, at the time of its arrival at $\mathcal{Q}_j$, a system with the same statistics as a system with one less customer.* Consider a product-form closed-queuing network with N customers. Let $\bar{A}_m(N)$ denote the average number of customers in queue $\mathcal{Q}_m$ seen by an arriving customer. Let $\bar{L}_m(N)$ denote the average length of queue $\mathcal{Q}_m$ in such a system. Then, the above result implies

$$\bar{A}_m(N) = \bar{L}_m(N-1) \qquad (1)$$

Given this result, the MVA algorithm iteratively computes the average response time of a request. The MVA algorithm uses Equation 1 to introduce customers into the queuing network, one by one, and determines the resulting average delays at various queues at each

---

[2]The term product-form applies to any queuing network in which the expression for the equilibrium probability has the form of $P(n_1, \cdots, n_M) = \frac{1}{G(N)} \pi_{i=1}^M f_i(n_i)$ where $f_i(n_1)$ is some function of the number of jobs at the ith queue, G(N) is a normalizing constant. Product form solutions are known to exist for a broad class of networks, including ones where the scheduling discipline at each queue is processor sharing (PS).

step. It terminates when all N customers have been introduced, and yields the average response time experienced by N concurrent customers. Note that a session in our model corresponds to a customer in the result described by Equation 1. The MVA algorithm for an M-tier Internet application servicing N sessions simultaneously is presented in Algorithm 1 and the associated notation is in Table 1.

The algorithm uses the notion of a *visit ratio* for each queue $\mathcal{Q}_1$, $\cdots$, $\mathcal{Q}_M$. The visit ratio V_m for queue $\mathcal{Q}_m$ ($1 \leq m \leq M$) is defined as the average number of visits made by a request to $\mathcal{Q}_m$ during its processing (that is, from when it emanates from $\mathcal{Q}_0$ and when it returns to it). Visit ratios are easy to compute from the transition probabilities $p_1, \cdots, p_M$ and provide an alternate representation of the queuing network. The use of visit ratios in lieu of transition probabilities enables the model to capture multiple visits to a tier regardless of whether they occur sequentially or in parallel—the visit ratio is only concerned with the mean number of visits made by a request to a queue and not *when* or in *what order* these visits occur.

Thus, given the average service times and visit ratios for the queues, the average think time of a session, and the number of concurrent sessions, the algorithm computes the average response time $\bar{R}$ of a request.

---

input : $N, \bar{S}_m, V_m, 1 \leq m \leq M; \bar{Z}$
output : $\bar{R}_m$ (avg. delay at $\mathcal{Q}_m$), $\bar{R}$ (avg. resp. time)
initialization:
$\bar{R}_0 = \bar{D}_0 = \bar{Z}; \bar{L}_0 = 0;$
for $m = 1$ *to* M **do**
    $\bar{L}_m = 0;$
    $\bar{D}_m = V_m \bar{S}_m$ /* service demand at each queue */;
end
/* introduce N customers, one by one */
for $n = 1$ *to* N **do**
    for $m = 1$ *to* M **do**
        $\bar{R}_m = \bar{D}_m(1 + \bar{L}_m)$ /* avg. delay at each que.*/;
    end
    $\tau = \left(\dfrac{n}{\bar{R}_0 + \sum_{m=1}^M \bar{R}_m} \right)$ /* throughput */;
    for $m = 1$ *to* M **do**
        $\bar{L}_m = \tau \cdot \bar{R}_m$ /* update queue lengths (little's law) */;
    end
    $\bar{L}_0 = \tau \cdot \bar{R}_0;$
end
$\bar{R} = \sum_{m=1}^{m=M} \bar{R}_m$ /* response time */;

Algorithm 1: Mean-value analysis algorithm for an M-tier application.

3.3 Estimating the Model Parameters

In order to compute the response time, the model requires several parameters as inputs. In practice, these parameters can be estimated by monitoring the application as it services its workload. To do so, we assume that the underlying operating system and application software components (such as the Apache Web server) provide monitoring hooks to enable accurate estimation of these parameters. Our experience with the Linux-based multi-tier applications used in our experiments is that such functionality is either already available or can be implemented at a modest cost. The rest of this section describes how the various model parameters can be estimated in practice.

Symbol	Meaning
M	Number of application tiers
N	Number of sessions
$\mathcal{Q}_m$	Queue representing tier T_m ($1 \leq m \leq M$)
$\mathcal{Q}_0$	Inf. server system to capture sessions
$\bar{Z}$	User think time
$\bar{S}_m$	Avg. per-request service time at $\mathcal{Q}_m$
$\bar{L}_m$	Avg. length of $\mathcal{Q}_m$
τ	Throughput
$\bar{R}_m$	Avg. per-request delay at $\mathcal{Q}_m$
$\bar{R}$	Avg. per-request response time
$\bar{D}_m$	Avg. per-request service demand at $\mathcal{Q}_m$
V_m	Visit ratio for $\mathcal{Q}_m$
$\bar{A}_m$	Avg. num. customers in $\mathcal{Q}_m$ seen by an arriving customer

Table 1: Notation used in describing the MVA algorithm.

Estimating visit ratios: The visit ratio for any tier of a multi-tier application is the average number of times that tier is invoked during a request's lifetime. Let λ_{req} denote the number of requests serviced by the entire application over a duration t. Then the visit ratio for tier T_i can be simply estimated as

$$V_i \approx \frac{\lambda_i}{\lambda_{req}}$$

where λ_i is the number of requests serviced by that tier in that duration. By choosing a suitably large duration t, a good estimate for V_i can be obtained. We note that the visit ratios are easy to estimate in an online fashion. The number of requests serviced by the application λ_{req} can be monitored at the application sentry. For replicated tiers, the number of requests serviced by all servers of that tier can be monitored at the dispatchers. Monitoring of both parameters requires simple counters at these components. For non-replicated tiers that lack a dispatcher, the number of serviced requests can be determined by real-time processing of the tier logs. In the database tier, for instance, the number of queries and transactions processed over a duration t can be determined by processing the database log using a script.

Estimating service times: Application components such as Web, Java, and database servers all support extensive logging facilities and can log a variety of useful information about each serviced request. In particular, these components can log the residence time of individual requests as observed at that tier—the residence time includes the time spent by the request at this tier and *all the subsequent tiers* that processed this request. This logging facility can be used to estimate per-tier service times. Let $\bar{X}_i$ denote the average per-request residence time at tier T_i. We start by estimating the mean service time at the last tier. Since this tier does not invoke services from any other tiers, the request execution time at this tier under lightly loaded conditions is an excellent estimate of the service time. Thus, we have,

$$\bar{S}_M \approx \bar{X}_M$$

Let S_i, X_i, and n_i be random variables denoting the service time of a request at a tier T_i, residence time of a request at tier T_i, and the number of times T_i requests service from T_{i+1} as part of the overall request processing, respectively. Then, under lightly loaded conditions,

$$S_i = X_i - n_i \cdot X_{i+1}, \ 1 \leq i < M.$$

Taking averages on both sides, we get,

$$\bar{S}_i = \bar{X}_i - E\left[n_i \cdot X_{i+1}\right]$$

Since n_i and X_{i+1} are independent, this gives us,

$$\bar{S}_i = \bar{X}_i - \bar{n}_i \cdot \bar{X}_{i+1} = \bar{X}_i - \left(\frac{V_{i+1}}{V_i}\right) \cdot \bar{X}_{i+1}$$

Thus, the service times at tiers $T_1, \cdots, T_{M-1}$ can be estimated.

Estimating think times: The average user think time, $\bar{Z}$, can be obtained by recording the arrival and finish times of individual requests at the sentry. $\bar{Z}$ is estimated as the average time elapsed between when a request finishes and when the next request (belonging to the same session) arrives at the sentry. By using a sufficient number of observations, we can obtain a good estimate of $\bar{Z}$.

Increased Service Times During Overloads: Our estimation of the tier-specific service times assumed lightly loaded conditions. As the load on a tier grows, software overheads such as waiting on locks, virtual memory paging, and context switch overheads, that are not captured by our model, can become significant components of the request processing time.

Incorporating the impact of increased context switching overhead or contention for memory or locks into our model is non-trivial. Rather than explicitly modeling these effects, we implicitly account for their impact by associating increased service times with requests under heavy loads. We use the Utilization Law [10] for a queuing system which states that $S = \rho/\tau$, where ρ and τ are the queue utilization and throughput, respectively. Consequently, we can improve our estimate of the average service time at tier T_i as

$$\bar{S'}_i = \max\left(\bar{S}_i, \frac{\rho_i}{\tau_i}\right)$$

where ρ_i is the utilization of the busiest resource (e.g. CPU, disk, or network interface) and τ_i is the tier throughput. Since all modern operating systems support facilities for monitoring system performance (e.g., the sysstat package in Linux [16]), the utilizations of various resources is easy to obtain online. Similarly, the tier throughput ρ_i can be determined at the dispatcher (or from logs) by counting the number of completed requests in a duration t.

4. MODEL ENHANCEMENTS

This section proposes enhancements to our baseline model to capture four application artifacts—replication and load imbalance at tiers, concurrency limits, and multiple session classes.

4.1 Replication and Load Imbalance at Tiers

Recall that our baseline model assumes a single server (queue) per tier and consequently does not support the notion of replication at a tier. We now enhance our model to handle this scenario. Let r_i denote the number of replicas at tier T_i. Our approach to capture replication at tier T_i is to replace the single queue $\mathcal{Q}_i$ with r_i queues, $\mathcal{Q}_i^1, \cdots, \mathcal{Q}_i^{r_i}$, one for each replica. A request in any queue can now make a transition to any of the r_{i-1} queues of the previous tier or to any of the r_{i+1} queues of the next tier.

In general, whenever a tier is replicated, a dispatcher is necessary to distribute requests to replicas. The dispatcher determines which request to forward to which replica and directly influences the transitions made by a request.

The dispatcher is also responsible for balancing load across replicas. In a perfectly load balanced system, each replica processes $\frac{1}{r_i}$ fraction of the total workload of that tier. In practice, however, perfect load balancing is difficult to achieve for the following reasons. First, if a session is stateful, successive requests will need to be serviced by the same stateful server at each tier; the dispatcher is forced

to forward all requests from a session to this replica regardless of the load on other replicas. Second, if caching is employed by a tier, a session and its requests may be preferentially forwarded to a replica where a response is likely to be cached. Thus, sessions may have *affinity* for particular replicas. Third, different sessions impose different processing demands. This can result in variability in resource usage of sessions, and simple techniques such as forwarding a new session to the least-loaded replica may not be able to counter the resulting load imbalance. Thus, the issues of replication and load imbalance are closely related. Our enhancement captures the impact of both these factors.

In order to capture the load imbalance across replicas, we explicitly model the load at individual replicas. Let λ_i^j denote the number of requests forwarded to the jth most loaded replica of tier T_i over some duration t. Let λ_i denote the total number of requests handled by that tier over this duration. Then, the imbalance factor β_i^j is computed as

$$\beta_i^j = \left(\frac{\lambda_i^j}{\lambda_i} \right)$$

We use exponentially smoothed averages of these ratios $\bar{\beta}_i^j$ as measures of the load imbalance at individual replicas. The visit ratios of the various replicas are then chosen as

$$V_i^j = V_i \bar{\beta}_i^j$$

The higher the load on a replica, the higher the value of the imbalance factor, and the higher its visit ratio. In a perfectly load-balanced system, $\beta_i^j = \frac{1}{r_i}, \forall j$. Observe that the number of requests forwarded to a replica λ_i^j and the total number of requests λ_i can be measured at the dispatcher using counters. The MVA algorithm can then be used with these modified visit ratios to determine the average response time.

4.2 Handling Concurrency Limits at Tiers

The software components of an Internet application have limits on the amount of concurrency they can handle. For instance, the Apache Web server uses a configurable parameter to limit the number of concurrent threads or processes that are spawned to service requests. This limit prevents the resident memory size of Apache from exceeding the available RAM and prevents thrashing. Connections are turned away when this limit is reached. Other tiers impose similar limits.

Our baseline model assumes that each tier can service an unbounded number of simultaneous requests and fails to capture the behavior of the application when the concurrency limit is reached at any tier. This is depicted in Figure 4(a), which shows the response time of a three-tier application called Rubis that is configured with a concurrency limit of 150 for the Apache Web server and a limit of 75 for the middle Java tier (details of the application appear in Section 5.1). As shown, the response times predicted by the model match the observed response times until the concurrency limit is reached. Beyond this point, the model continues to assume an increasing number of simultaneous requests being serviced and predicts an increase in response time, while the actual response time of successful requests shows a flat trend due to an increasing number of dropped requests.

In general, when the concurrency limit is reached at tier T_i, one of two actions are possible: (1) the tier can silently drop additional requests and rely upon a timeout mechanism in tier T_{i-1} to detect these drops, or (2) the tier can explicitly notify tier T_{i-1} of its inability to serve the request (by returning an error message). In either case, tier T_{i-1} may reissue the request some number of times before abandoning its attempts. It will then either drop the request or ex-
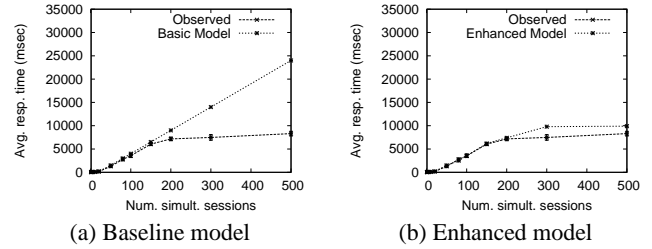

(a) Baseline model (b) Enhanced model

Figure 4: Response time of Rubis with 95% confidence intervals. A concurrency limit of 150 for Apache and 75 for the Java servlet tier is used. Figure (a) depicts the deviation of the baseline model from observed behavior when concurrency limit is reached. Figure (b) depicts the ability of the enhanced model to capture this effect.

plicitly notify its preceding tier. Finally, tier T_1 can notify the client of the failure.

Rather than distinguishing these possibilities, we employ a general approach for capturing these effects. Let K_i denote the concurrency limit at $\mathcal{Q}_i$. To capture requests that are dropped at tier T_i when its concurrency limit is reached, we add additional transitions, one for each queue representing a tier, to the basic model that we presented in Figure 3. At the entrance of $\mathcal{Q}_i$, we add a transition into an infinite server queuing subsystem $\mathcal{Q}_i^{drop}$. Let p_i^{drop} denote the probability of a request transiting from $\mathcal{Q}_{i-1}$ to $\mathcal{Q}_i^{drop}$ as shown in Figure 5. $\mathcal{Q}_i^{drop}$ has a mean service time of X_i^{drop}. This enhancement allows us to distinguish between the processing of requests that get dropped due to concurrency limits and those that are processed successfully. Requests that are processed successfully are modeled exactly as in the basic model. Requests that are dropped at tier T_i experience some delay in the subsystem $\mathcal{Q}_i^{drop}$ before returning to $\mathcal{Q}_0$—this models the delay between when a request is dropped at tier T_i and when this information gets propagated to the client that initiated the request.

Like in the baseline model, we can use the MVA algorithm to compute the response time of a request. The algorithm computes the fraction of requests that finish successfully and those that encounter failures, as well as the delays experienced by both types of requests. To do so, we need to estimate the additional parameters that we have added to our basic model, namely, p_i^{drop} and X_i^{drop} for each tier T_i.

Estimating p_i^{drop}: Our approach to estimate p_i^{drop} consists of the following two steps.

Step 1 : *Estimate throughput of the queuing network if there were no concurrency limits*: Solve the queuing network shown in Figure 5 using the MVA algorithm using $p_i^{drop} = 0$ (i.e., assuming that the queues have no concurrency limits). Let λ denote the throughput computed by the MVA algorithm in this step.

Step 2 : *Estimate p_i^{drop}*: Treat $\mathcal{Q}_i$ as an open, finite-buffer M/M/1/K_i queue with arrival rate λV_i (using the λ computed in Step 1). Estimate p_i^{drop} as the probability of buffer overflow in this M/M/1/K_i queue [8].

Estimating X_i^{drop}: An estimate of X_i^{drop} is application-specific and depends on the manner in which information about dropped requests is conveyed to the client, and how the client responds to it. In our current model we make the simplifying assumption that upon detecting a failed request, the client reissues the request. This is captured by the transitions from $\mathcal{Q}_i^{drop}$ back to $\mathcal{Q}_0$ in Figure 5. Our
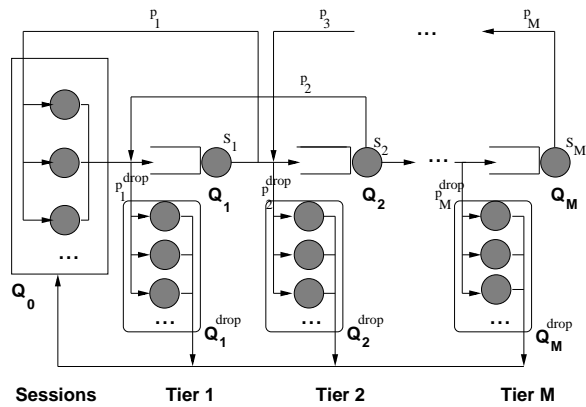
Figure 5: Multi-tier application model enhanced to handle concurrency limits.
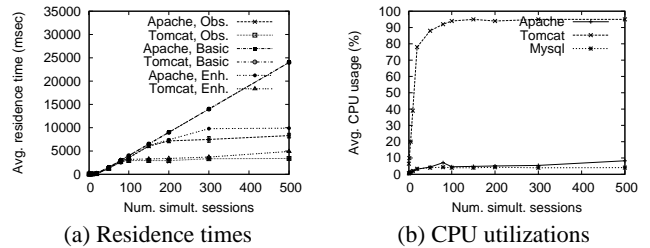

(a) Residence times   (b) CPU utilizations

Figure 6: Rubis based on Java servlets: bottleneck at CPU of middle tier. The concurrency limits for the Apache Web server and the Java servlets container were set to be 150 and 75, respectively.

approach for estimating X_i^{drop} is to subject the application to an offline workload that causes the limit to be exceeded only at tier T_i (this can be achieved by setting a low concurrency limit at that tier and sufficiently high limits at all the other tiers), and then record the response times of the requests that do not finish successfully. X_i^{drop} is then estimated as the difference between the average response time of these unsuccessful requests and the sum of the service times at tiers $T_1, \cdots, T_{i-1}$.

In Figure 4(b) we plot the response times for Rubis as predicted by our enhanced model. We find that this enhancement enables us to capture the behavior of the Rubis application even when its concurrency limit is reached.

4.3 Handling Multiple Session Classes

Internet applications typically classify incoming sessions into multiple *classes*. To illustrate, an online brokerage Web site may define three classes and may map financial transactions to the *Gold* class, customer requests such as balance inquiries to the *Silver* class, and casual browsing requests from non-customers to the *Bronze* class. Typically such classification helps the application sentry to preferentially admit requests from more important classes during overloads and drop requests from less important classes.

We can extend our baseline model to account for the presence of different session classes and to compute the response time of requests within each class. Consider an Internet application with L session classes: $C_1, C_2, \ldots, C_L$. Assume that the sentry implements a classification algorithm to map each incoming session to one of these classes. We can use a straightforward extension of the MVA algorithm to deal with multiple session classes. Due to lack of space, we omit specific details. We note that this algorithm requires the visit ratios, service times, and think time to be measured on a *per-class basis*. Given a L-tuple $(N_1, \cdots, N_L)$ of sessions belonging to the L classes that are simultaneously serviced by the application, the algorithm can compute the average delays incurred at each queue and the end-to-end response time on a per-class basis. In Section 6.2 we discuss how this algorithm can be used to flexibly implement session policing policies in an Internet application.

4.4 Other Enhancements and Salient Features

Our closed queuing model has several desirable features.

Simplicity: For an M-tier application with N concurrent sessions, the MVA algorithm has a time complexity of $O(MN)$. The algorithm is simple to implement, and as argued earlier, the model parameters are easy to measure online.

Generality: Our model can handle an application with arbitrary number of tiers. Further, when the scheduling discipline is processor sharing (PS), the MVA algorithm works without making any assumptions about the service time distributions of the customers [10]. This feature is highly desirable for two reasons: (1) it is representative of scheduling policies in commodity operating systems (e.g., Linux's CPU time-sharing), and (2) it implies that our model is sufficiently general to handle workloads with an arbitrary service time requirements.[3]

While our model is able to capture a number of application idiosyncrasies, certain scenarios are not explicitly captured.

Multiple resources: We model each server occupied by a tier using a single queue. In reality, the server contains various resources such as the CPU, disk, memory, and the network interface. Our model currently does not capture the utilization of various server resources by a request at a tier. An enhancement to the model where various resources within a server are modeled as a network of queues is the subject of future work.

Resources held simultaneously at multiple tiers: Our model essentially captures the passage of a request through the tiers of an application as a juxtaposition of periods, during each of which the request utilizes the resources at *exactly one* tier. Although this is a reasonable assumption for a large class of Internet applications, it does not apply to certain Internet applications such as streaming video servers. A video server that is constructed as a pipeline of processing modules will have *all* of its modules or "tiers" active as it continuously processes and streams a video to a client. Our model does not apply to such applications.

5. MODEL VALIDATION

In this section we present our experimental setup followed by our experimental validation of the model.

5.1 Experimental Setup

Applications: We use two open-source multi-tier applications in our experimental study. *Rubis* implements the core functionality of an eBay like auction site: selling, browsing, and bidding. It implements three types of user sessions, has nine tables in the database and defines 26 interactions that can be accessed from the clients' Web browsers. *Rubbos* is a bulletin-board application modeled after

---

[3]The applicability of the MVA algorithm is more restricted with some other scheduling disciplines. E.g., in the presence of a FIFO scheduling discipline at a queue, the service time at a queue needs to be exponentially distributed for the MVA algorithm to be applicable.

(a) Response time   (b) Residence times   (c) CPU utilizations
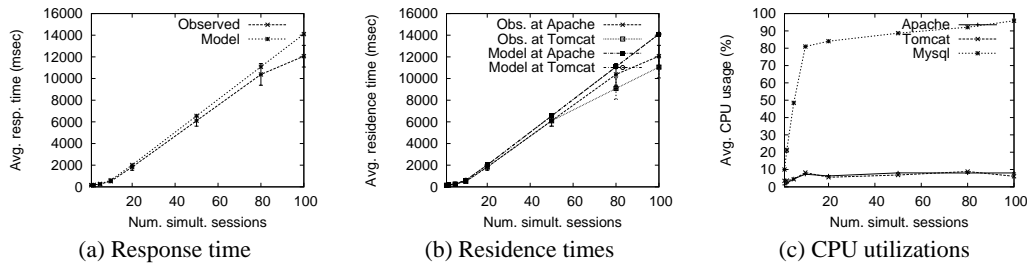
Figure 7: Rubis based on Java servlets: bottleneck at CPU of database tier. The concurrency limits for the Apache Web server and the Java servlets container were set to be 150 and 75, respectively.

an online news forum like Slashdot. Users have two different levels of access: regular user and moderator. The main tables in the database are the users, stories, comments, and submissions tables. Rubbos provides 24 Web interactions. Both applications were developed by the DynaServer group at Rice University [5]. Each application contains a Java-based client that generates a session-oriented workload. We modified these clients to generate the workloads and take the measurements needed by our experiments. We chose an average duration of 5 min for the sessions of both Rubis and Rubbos. For both applications, the think time was chosen from an exponential distribution with a mean of 1 sec.

We used 3-tier versions of these applications. The front tier was based on Apache 2.0.48 Web server. We experimented with two implementations of the middle tier for Rubis—(i) based on Java servlets, and (ii) based on Sun's J2EE Enterprise Java Beans (EJBs). The middle tier for Rubbos was based on Java servlets. We employed Tomcat 4.1.29 as the servlets container and JBoss 3.2.2 as the EJB container. We used *Kernel TCP Virtual Server* (ktcpvs) version 0.0.14 [9] to implement the application sentry. ktcpvs is an open-source, Layer-7 request dispatcher implemented as a Linux kernel module. A round-robin load balancer implemented in ktcpvs was used for Apache. Request dispatching for the middle tier was performed by *mod_jk*, an Apache module that implements a variant of round robin request distribution while taking into account session affinity. Finally, the database tier was based on the Mysql 4.0.18 database server.

Hosting environment: We conducted experiments with the applications hosted on two different kinds of machines. The first hosting environment consisted of IBM servers (model 6565-3BU) with 662 MHz processors and 256MB RAM connected by 100Mbps ethernet. The second setting, used for experiments reported in Section 6, had Dell servers with 2.8GHz processors and 512MB RAM interconnected using gigabit ethernet. This served to verify that our model was flexible enough to capture applications running on different types of machines. Finally, the workload generators were run on machines with Pentium-III processors with speeds 450MHz-1GHz and RAM sizes in the range 128-512MB. All the machines ran the Linux 2.4.20 kernel.

5.2 Performance Prediction

We conduct a set of experiments with the purpose of ascertaining the ability of our model to predict the response time of multi-tier applications. We experiment with (i) two kinds of applications (Rubis and Rubbos), (ii) two different implementations of Rubis (based on Java servlets and EJBs), and (iii) different workloads for Rubis. Each of the three application tiers are assigned one server except in the experiments reported in Section 5.4. We vary the number of concurrent sessions seen by the application and measure the average re-

sponse times of successfully finished requests over 30 sec intervals. Each experiment lasts 30 min. We compute the average response time and the 95% confidence intervals from these observations.

Our first experiment uses Rubis with a Java servlets-based middle tier. We use two different workloads—$W1$: CPU-intensive on the Java servlets tier, and $W2$: CPU-intensive on the database tier. These were created by modifying the Rubis client so that it generated an increased fraction of requests that stressed the desired tier. Earlier, in Figure 4(b) we had presented the average response time and 95% confidence intervals for sessions varying from 1 to 500 for the workload $W1$. Also plotted were the average response times predicted by our basic model and our model enhanced to handle concurrency limits. Additionally, we present the observed and predicted residence times in Figure 6(a). Figure 6(b) shows that the CPU on the Java servlets tier becomes saturated beyond 100 sessions for this workload. As already explained in Section 4.2, the basic model fails to capture the response times for workloads higher than about 100 sessions due to an increase in the fraction of requests that arrive at the Apache and servlets tiers only to be dropped because of the tiers operating at their concurrency limits. We find that our enhanced model is able to capture the effect of dropped requests at these high workloads and continues to predict response times well for the entire workload range.

Figure 7 plots the response times, the residence times, and the server CPU utilizations for servlets-based Rubis subjected to the workload $W2$ with varying number of sessions. As shown in Figure 7(c), the CPU on the database server is the bottleneck resource for this workload. We find that our basic model captures response times well. The predicted response times are within the 95% confidence interval of the observed average response time for the entire workload range.

Next, we repeat the experiment described above with Rubis based on an EJB-based middle tier. Our results are presented in Figure 8. Again, our basic model captures the response time well until the concurrency limits at Apache and JBoss are reached. As the number of sessions grows beyond this point, increasingly large fractions of requests are dropped, the request throughput saturates, and the response time of requests that finish successfully shows a flat trend. Our enhancement to the model is again found to capture this effect well.

Finally, we repeat the above experiment with the Rubbos application. We use a Java servlets based middle tier for Rubbos and subject the application to the workload $W1$ that is CPU-intensive on the servlets tier. Figure 9 presents the observed and predicted response times as well as the server CPU utilizations. We find that our enhanced model predicts response times well over the chosen workload range for Rubbos.
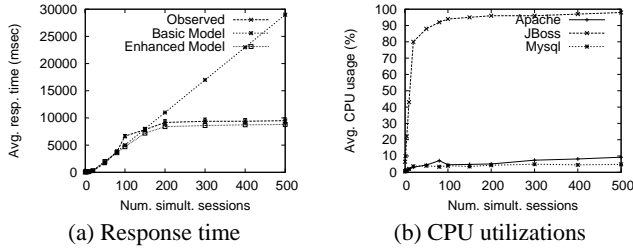
(a) Response time   (b) CPU utilizations

Figure 8: Rubis based on EJB: bottleneck at CPU of middle tier. The concurrency limits for the Apache Web server and the Java servlets container were set to be 150 and 75, respectively.

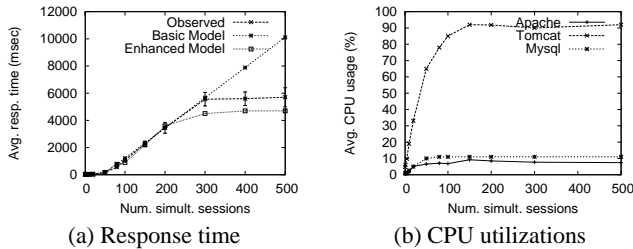(a) Response time   (b) CPU utilizations

Figure 9: Rubbos based on Java servlets: bottleneck at CPU of middle tier. The concurrency limits for the Apache Web server and the Java servlets container were set to be 150 and 75, respectively.

5.3 Query Caching at the Database

Recent versions of the Mysql server feature a query cache. When in use, the query cache stores the text of a SELECT query together with the corresponding result that was sent to the client. If the identical query is received later, the server retrieves the results from the query cache rather than parsing and executing the query again. Query caching at the database has the effect of reducing the average service time at the database tier. We conduct an experiment to determine how well our model can capture the impact of query caching on response time. We subject Rubbos to a workload consisting of 50 simultaneous sessions. To simulate different degrees of query caching at Mysql, we use a feature of Mysql queries that allows the issuer of a query to specify that the database server not use its cache for servicing this query[4]. We modified the Rubbos servlets to make them request different fractions of the queries with this option. For each degree of caching we plot the average response time with 95% confidence intervals in Figure 10. As expected, the observed response time decreases steadily as the degree of query caching increases—the average response time is nearly 1400 msec without query caching and reduces to about 100 msec when all the queries are cached. In Figure 10 we also plot the average response time predicted by our model for different degrees of caching. We find that our model is able to capture well the impact of the reduced query processing time with increasing degrees of caching on average response time. The predicted response times are found to be within the 95% confidence interval of the observed response times for the entire range of query caching.

5.4 Load Imbalance at Replicated Tiers

---

[4]Specifically, replacing a `SELECT` with `SELECT SQL_NO_CACHE` ensures that Mysql does not cache this query.
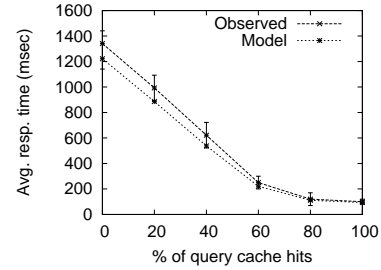

Figure 10: Caching at the database tier of Rubbos.

We configure Rubis using a replicated Java servlets tier—we assign three servers to this tier. We use the workload $W1$ with 100 simultaneous sessions. The user think times for a session are chosen using an exponential distribution whose mean is chosen uniformly at random from the set $\{1\ sec, 5\ sec\}$. We choose short-lived sessions with a mean session duration of 1 minute. Our results are presented in Figure 11. Note that replication at the middle tier causes the response times to be significantly smaller than in the experiment depicted in Figure 6(a). Further, choosing sessions with two widely different think times ensures variability in the workload imposed by individual sessions and creates load imbalance at the middle tier.

Figure 11(a) plots the number of requests passing through each of the three servers in the servlets tier over 30 sec intervals during a 10 min run of this experiment; Figure 11(b) plots the average end-to-end response times for these requests. These figures show the imbalance in the load on the three replicas. Also, the most loaded server changes over time—choosing a short session duration causes the load imbalance to shift among replicas frequently. Figure 11(c) plots the average response times observed for requests passing through the three servers—instead of presenting response times corresponding to specific servers, we plot values for the least loaded, the second least loaded, and the most loaded server. Figure 11(c) also shows the response times predicted by the model assuming *perfect load balancing* at the middle tier. Under this assumption, we see a deviation between the predicted values and the observed response times. Next, we use the model enhancement described in Section 4.1 to capture load imbalance. For this workload the values for the load imbalance factors used by our enhancement were determined to be $\bar{\beta}_2^1 = 0.25$, $\bar{\beta}_2^2 = 0.32$, and $\bar{\beta}_2^3 = 0.43$. We plot the response times predicted by the enhanced model at the extreme right in Figure 11(c). We observe that the use of these additional parameters improves our prediction of the response time. The predicted average response time (1350 msec) closely matched the observed value (1295 msec); with the assumption of perfect load balancing the model underestimated the average response time to be 950 msec.

5.5 Multiple Session Classes

We created two classes of Rubis sessions using the workloads $W1$ and $W2$ respectively. Recall that the requests in these classes have different service time requirements at different tiers—$W1$ is CPU-intensive on the Java servlets tier while $W2$ is CPU intensive on the database tier. We conduct two sets of experiments, each of which involves keeping the number of sessions of one class fixed at 10 and varying the number of sessions of the other class. We then compute the per-class average response time predicted by the multi-class version of our model (Section 4.3). We plot the observed and predicted response times for the two classes in Figure 12. While the predicted response times closely match the observed values for the first experiment, in the second experiment (Figure 12(b)), we ob-

(a) Num. requests (per-replica)　(b) Resp. times (per-replica)　(c) Resp. times (based on load)
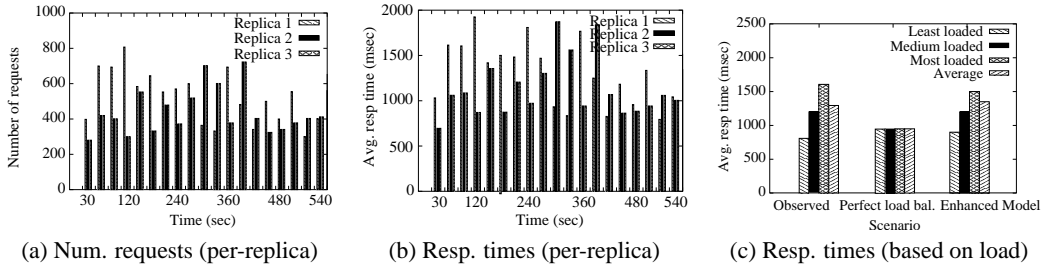
Figure 11: Load imbalance at the middle tier of Rubis. (a) and (b) present number of requests and response times classified on a per-replica basis; (c) presents response times classified according to most loaded, second most loaded, and most loaded replicas and overall average response times.
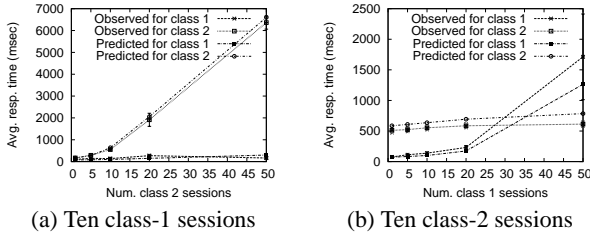

(a) Ten class-1 sessions　(b) Ten class-2 sessions

Figure 12: Rubis serving sessions of two classes. Sessions of class 1 were generated using workload $W1$ while those of class 2 were generated using workload $W2$.

serve that our model underestimates the response time for class 1 for 50 sessions—we attribute this to an inaccurate estimation of the service time of class 1 requests at the servlets tier at this load.

6. APPLICATIONS OF THE MODEL

In this section we demonstrate some applications of our model for managing resources in a data center. We also discuss some important issues related to the online use of our model.

6.1 Dynamic Capacity Provisioning and Bottleneck Identification

Dynamic capacity provisioning is a useful technique for handling the multi-time-scale variations seen in Internet workloads. The goal of dynamic provisioning is to dynamically allocate sufficient capacity to the tiers of an application so that its response time needs can be met even in the presence of the peak workload. Two key components of a dynamic provisioning technique are: (i) predicting the workload of an application, and (ii) determining the capacity needed to serve this predicted workload. The former problem has been addressed in papers such as [6]. The workload estimates made by such predictors can be used by our model to address the issue of how much capacity to provision. Observe that *the inputs to our model-based provisioning technique are the workload characteristics, number of sessions to be serviced simultaneously, and the response time target, and the desired output is a capacity assignment for the application*. We start with an initial assignment of one server to each tier. We use the MVA algorithm to determine the resulting average response time as described in Sections 3 and 4. In case this is worse than the target, we use the MVA algorithm to determine, for each replicable tier, the response time resulting from the addition of one more server to it. We add a server to the tier that results in the most improvement in response time. We repeat this till we have an assignment for which the

predicted response time is below the target—this assignment yields the capacity to be assigned to the application's tiers[5]. The above provisioning procedure has a time complexity of $O(kMN)$, where k is the number of servers that the application is eventually assigned, M is the the number of tiers, and N is the number of sessions. Since provisioning decisions are typically made over periods of tens of minutes or hours, this overhead is practically feasible.

We conduct an experiment to demonstrate the application of our model to dynamically provision Rubis configured using Java servlets at its middle tier. We assume an idealized workload predictor that can accurately forecast the workload for the near future. We generated a 1-hour long session arrival process based on a Web trace from the 1998 Soccer World Cup site [2]; this is shown in Figure 13(a). Sessions are generated according to this arrival process using workload $W1$.

We implemented a provisioning unit that invokes the model-based procedure described above every 10 min to determine the capacity required to handle the workload during the next interval. Our goal was to maintain an average response time of 1 sec for Rubis requests. Since our model requires the number of simultaneous sessions as input, the provisioning unit converted the peak rate during the next interval into an estimate of the number of simultaneous sessions for which to allocate capacity using Little's Law [8] as $N = \Lambda \cdot d$, where Λ is the peak session arrival rate during the next interval as given by the predictor and d is the average session duration. The provisioning unit ran on a separate server. It implemented scripts that remotely log on to the application sentry and the dispatchers for the affected tiers after every re-computation to enforce the newly computed allocations. The concurrency limits of the Apache Web server and the Tomcat servlets container were both set to 100. We present the working of our provisioning unit and the performance of Rubis in Figure 13(b). The provisioning unit is successful in changing the capacity of the servlets tier to match the workload—recall that workload $W1$ is CPU intensive on this tier. The session arrival rate goes up from about 10 sess/min at $t = 20$ min to nearly 30 sess/min at $t = 40$ min. Correspondingly, the request arrival rate increases from about 1500 req/min to about 4200 req/min. The provisioning unit increases the number of Tomcat replicas from 2 to a maximum of 7 during the experiment. Further, at $t = 30$ min, the number of simultaneous sessions during the upcoming 10 min interval is predicted to be higher than the concurrency limit of the Apache tier.

---

[5]Note that our current discussion assumes that it is always possible to meet the response time target by adding enough servers. Sometimes this may not be possible (e.g., due to the workload exceeding the entire available capacity, or a non-replicable tier becoming saturated) and we may have to employ admission control in addition to provisioning. This is discussed in Section 6.2.
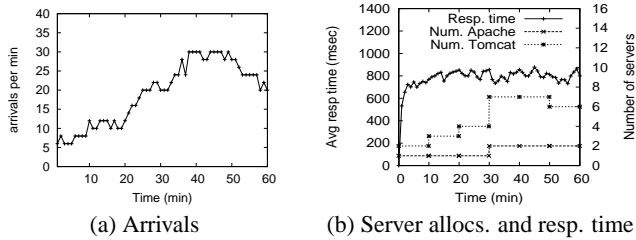
(a) Arrivals   (b) Server allocs. and resp. time

Figure 13: Model-based dynamic provisioning of servers for Rubis.

(a) Model-based policing   (b) Policer admits more than capacity

Figure 14: Maximizing revenue via differentiated session policing in Rubis. The application serves two classes of sessions.

To prevent new sessions being dropped due to the connection limit being reached at Apache, a second Apache server is added to the application. Thus, our model-based provisioning is able to identify potential bottlenecks at different tiers (connections at Apache and CPU at Tomcat) and maintain response time targets by adding capacity appropriately. We note that the single-tier models described in Section 2.3 will only be able to add capacity to one tier and will fail to capture such changing bottlenecks.

6.2 Session Policing and Class-based Differentiation

Internet applications are known to experience unexpected surges in their workload, known as *flash crowds* [20]. Therefore an important component of any such application is a sentry that polices incoming sessions to an application's server pool—incoming sessions are subjected to admission control at the sentry to ensure that the contracted performance guarantees are met; excess sessions are turned away during overloads. In an application supporting multiple classes of sessions, with possibly different response time requirements and revenue schemes for different classes, it is desirable to design a sentry that, during a flash crowd, can determine a subset of sessions admitting which would optimize a meaningful metric. An example of such a metric could be the overall expected revenue generated by the admitted sessions while meeting their response time targets (this constraint on response times will be assumed to hold in the rest of our discussion without being stated). Formally, given L session classes, $C_1, \cdots, C_L$, with up to N_i sessions of class C_i and using overall revenue as the metric to be optimized, the goal of the sentry is to determine an L-tuple $(N_1^{admit}, \cdots, N_L^{admit})$ such that

$$\forall n_i \leq N_i (1 \leq i \leq L), \ \sum_i rev_i(N_i^{admit}) \geq \sum_i rev_i(n_i)$$

where $rev_i(n_i)$ denotes the revenue generated by n_i admitted sessions of C_i.

Our multi-class model described in Section 4.3 provides a flexible procedure for realizing this. First observe that *the inputs to this procedure are the workload characteristics of various classes and the capacity assigned to the application tiers, and the desired output is the number of sessions of each class to admit*. In theory, we could use the multi-class MVA algorithm to determine the revenue yielded by every admissible L-tuple. Clearly this would be computationally prohibitive. Instead, we use a heuristic that considers the session classes in a non-increasing order of their revenue-per-session. For the class under consideration, it adds sessions till either all available sessions are exhausted, or adding another session would cause the response time of at least one class, as predicted by the model, to violate its target. The outcome of this procedure is an L-tuple of the number of sessions that can be used by the policer to make admission control decisions.
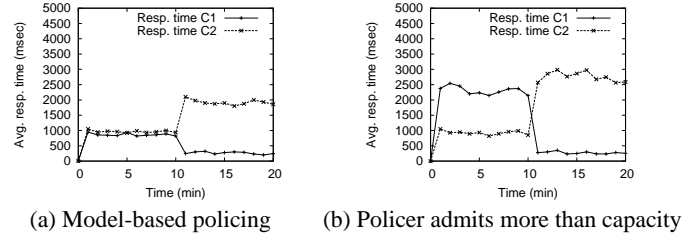
We now describe our experiments to demonstrate the working of the session policer for Rubis. We configured the servlets version of Rubis with 2 replicas of the servlets tier. Similar to Section 4.3, we chose $W1$ and $W2$ to construct two session classes C_1 and C_2 respectively. The response time targets for the two classes were chosen to be 1 sec and 2 sec; the revenue yielded by each admitted session was assumed to be \$0.1 and \$1 respectively. We assume session durations of exactly 10 min for illustrative purposes. We create the following flash crowd scenarios. We assume that 150 sessions of C_1 and 10 sessions of C_2 arrive at $t = 0$; 50 sessions each of C_1 and C_2 are assumed to arrive at $t = 10$ min. Figure 14(a) presents the working of our model-based policer. At $t = 0$, based on the procedure described above, the policer first admits all 10 sessions of the class with higher revenue-per-session, namely C_2; it then proceeds to admit as many sessions of C_1 as it can (90) while keeping the average response times under target. At $t = 10$ min, the policer first admits as many sessions of C_2 as it can (21); it then admits 5 sessions of C_1—admitting more would, according to the model, cause the response time of C_2 to be violated. We find from Figure 14(a) that the response time requirements of both the classes are met during the experiment. We make two additional observations: (i) during [0, 10] min, the response time of C_2 is well below its target of 2 sec—this is because there are only 10 sessions of this class, less than the capacity of the database tier for the desired response time target; since the 90 sessions of C_1 stress mainly the servlets tier (recall the nature of $W1$ and $W2$), they have minimal impact on the response time of C_2 sessions, which mainly exercise the database tier, and (ii) during (10, 20] min, the response time of C_1 is well below its target of 1 sec—this is because the policer admits only 5 C_1 sessions; the servlets tier is lightly loaded since the C_2 sessions do not stress it, and therefore the C_1 sessions experience low response times.

Figure 14(b) demonstrates the impact of admitting more sessions on application response time. At $t = 0$, the policer admits excess C_1 sessions—it admits 140 and 10 sessions respectively. We find that sessions of C_1 experience degraded response times (in excess of 2 sec as opposed to the desired 1 sec). Similarly, at $t = 10$ min, it admits excess C_2 sessions—it admits 5 and 31 sessions respectively. Now sessions of C_2 experience response time violations. Observe that admitting excess sessions of one class does not cause a perceptible degradation in the performance of the other class because they exercise different tiers of the application.

6.3 Overheads and Accuracy of Parameter Estimation

In this section we discuss issues related to the online use of our model.

System Overheads: The model requires several parameters to be measured at the sentry, the dispatchers, the application tiers, and in the operating system. In order to quantify the overheads imposed
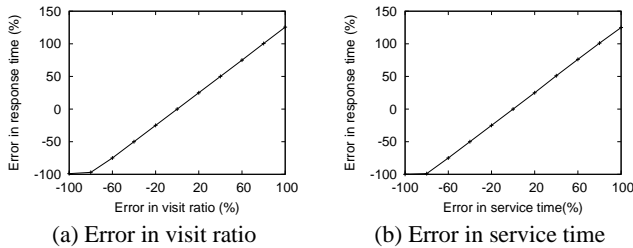
| (a) Error in visit ratio | (b) Error in service time |

Figure 15: Impact of error in estimating the model parameters.

by these online measurements, we monitored the CPU and network overheads imposed by the online monitoring components. Measurements on our hosting environment indicate that the CPU overhead of online parameter estimation is negligible—consistently less than 1% in a number of experiments. Similarly, real-time parsing and processing of application logs imposes a negligible overhead. The statistics measured at various components need to be periodically communicated to the sever running our model. We measured the network overhead of these messages. For Rubis servicing 500 concurrent sessions the network traffic created by our online estimation was about 0.5 Mbps, which is negligible on a gigabit ethernet LAN.

Inaccuracies in Parameter Estimation: To understand the impact of inaccuracies in the estimated parameters on the predicted response times, we deliberately introduced varying amounts of error in the estimated visit ratios and the estimated service times of the middle tier. We then used these parameter values to compute the mean response time using the model and determined the difference between the predicted response times with and without the errors. Figures 15(a) and (b) plot the error in average response time caused by varying degrees of error for Rubis servicing workload $W1$. Our results suggest a *linear dependence* between errors in the estimated parameters and those in predicted response time. This indicates that small errors in our measurements will only introduce small errors in the model predictions.

7. CONCLUSIONS

In this paper we presented an analytical model for multi-tier Internet applications. Our model is based on using a network of queues to represent how the tiers in a multi-tier application cooperate to process requests. Our model is (i) general enough to capture Internet applications with an arbitrary number of heterogeneous tiers, (ii) is inherently designed to handle session-based workloads, and (iii) can account for application idiosyncrasies such as load imbalances within a replicated tier, caching effects, the presence of multiple classes of sessions, and limits on the amount of concurrency at each tier. The model parameters are easy to measure and update. We validated the model using two open-source multi-tier applications running on a Linux-based server cluster. Our experiments demonstrated that our model faithfully captures the performance of these applications for a variety of workloads and configurations. We demonstrated the utility of our model in managing resources for Internet applications under varying workloads and shifting bottlenecks. As part of future work, we plan to investigate the suitability our model for capturing more diverse workloads (e.g., IO-intensive at certain tiers) and to design enhancements to handle these. Another direction is to extend our model to handle other kinds of scheduling disciplines (such as proportional-share scheduling) at the application servers.

8. REFERENCES

[1] T. Abdelzaher, K. G. Shin, and N. Bhatti. Performance Guarantees for Web Server End-Systems: A Control-Theoretical Approach. *IEEE Transactions on Parallel and Distributed Systems*, 13(1), Jan. 2002.

[2] M. Arlitt and T. Jin. Workload Characterization of the 1998 World Cup Web Site. Technical Report HPL-1999-35R1, HP Labs, 1999.

[3] A. Chandra, W. Gong, and P. Shenoy. Dynamic Resource Allocation for Shared Data Centers Using Online Measurements. In *Proceedings of Eleventh International Workshop on Quality of Service (IWQoS 2003)*, June 2003.

[4] R. Doyle, J. Chase, O. Asad, W. Jin, and A. Vahdat. Model-Based Resource Provisioning in a Web Service Utility. In *Proceedings of the 4th USITS*, Mar. 2003.

[5] Dynaserver project. http://compsci.rice.edu/CS/Systems/DynaServer/.

[6] J. Hellerstein, F. Zhang, and P. Shahabuddin. An Approach to Predictive Detection for Service Management. In *Proceedings of the IEEE Intl. Conf. on Systems and Network Management*, 1999.

[7] A. Kamra, V. Misra, and E. Nahum. Yaksha: A Controller for Managing the Performance of 3-Tiered Websites. In *Proceedings of the 12th IWQoS*, 2004.

[8] L. Kleinrock. *Queueing Systems, Volume 1: Theory*. John Wiley and Sons, Inc., 1975.

[9] Kernel TCP Virtual Server. http://www.linuxvirtualserver.org/software/ktcpvs/ktcpvs.html.

[10] E. Lazowska, J. Zahorjan, G. Graham, and K. Sevcik. *Quantitative System Performance*. Prentice–Hall, 1984.

[11] R. Levy, J. Nagarajarao, G. Pacifici, M. Spreitzer, A. Tantawi, and A. Youssef. Performance Management for Cluster Based Web Services. In *IFIP/IEEE Eighth International Symposium on Integrated Network Management*, volume 246, pages 247–261, 2003.

[12] D. Menasce. Web Server Software Architectures. In *IEEE Internet Computing*, volume 7, November/December 2003.

[13] Oracle9i. http://www.oracle.com/technology/products/oracle9i.

[14] S. Ranjan, J. Rolia, H. Fu, and E. Knightly. Qos-driven server migration for internet data centers. In *Proceedings of the Tenth International Workshop on Quality of Service (IWQoS 2002)*, May 2002.

[15] M. Reiser and S. Lavenberg. Mean-Value Analysis of Closed Multichain Queuing Networks. In *Journal of the Association for Computing Machinery*, volume 27, pages 313–322, 1980.

[16] Sysstat package. http://freshmeat.net/projects/sysstat.

[17] L. Slothouber. A Model of Web Server Performance. In *Proceedings of the 5th International World Wide Web Conference*, 1996.

[18] B. Urgaonkar and P. Shenoy. Cataclysm: Handling Extreme Overloads in Internet Services. In *Proceedings of the 23rd Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, July 2004.

[19] D. Villela, P. Pradhan, and D. Rubenstein. Provisioning Servers in the Application Tier for E-commerce Systems. In *Proceedings of the 12th IWQoS*, June 2004.

[20] M. Welsh and D. Culler. Adaptive Overload Control for Busy Internet Servers. In *Proceedings of the 4th USITS*, March

2003.